Assignment 1: Sequential Program Optimization

Louis Thomas

30 January 2018

1 Problem Description

The goal of this assignment is to take some intentionally slow code and optimize it. The given code (written by Ross Walker) was purposefully written to let students take the code and, between using compiler optimizations and hand-written ones, make the resulting executable process the same data at a much faster pace while producing the same or "same-enough" result.

The computations performed by the computer were modeled after a fictional function:

$$\sum_{j<i} \left(\frac{e^{(r_{i,j} * q_i)} * e^{(r_{i,j} * q_j)}}{r_{i,j}} - \frac{1}{a} \right) \text{ if } r_{i,j} \leq cut$$

where i and j are some particle, $r_{i,j}$ is the distance from particles i and j, cut is the cut-off distance between two particles, q_i is the energy scalar of particle i, and a is a constant.

The specific portion of the program that can benefit from optimization is a doubly-nested for-loop. Notable functions in the loop include $exp()$ and $sqrt()$.

2 Approach

The first optimization was to reduce a number of subtractions done within array indices on each loop iteration. The original loop counted from one to the number of atoms, however, 1 was subtracted from i and j each time they were used inside the loop. The change included starting i and j from zero and changing the $\leq$ to a $<$, and removing all the -1's from the array indices.

Secondly, an if statement including a check for $i < j$ was removed and instead incorporated into the inner loop. This cuts about half of the iterations from only checking that $(j < i)$ is false and continuing.

Third, a call to the square root function was moved until later. The original loop calculated $r_{i,j}$ in its entirety each iteration. This meant that, even if the value isn't included in the final sum, it was still calling the square root on it. Instead of comparing $r_{i,j}$ to the cutoff, $r_{i,j}^2$ is compared to cut^2. This required the small cost of calculating cut^2, but it also meant that $sqrt(r_{i,j}^2)$ was pushed back until it was necessary to be calcualted.

Fourth, many dereferences were moved out of the inner-most loop. This causes some dereferences to only happen 1 or n times, as opposed to up to $\frac{1}{2}n^2$ times. The arrays that were acted on included the *coords* and q arrays.

Fifth, the multiplying two exponents of the same base can be simplified to one exponentiation. From the original formula: $e^{r_{i,j}*q_i} * e^{r_{i,j}*q_j}$ can be simplified to $e^{r_{i,j}*q_i+r_{i,j}*q_j}$, and then $e^{r_{i,j}*(q_i+q_j)}$. An optimization in the code looks much like the previous simplification, thus not only reducing the number of function calls but also cutting the number of exponentiations by half.

A simple optimization follows from a mathematical characteristic in sums. Specifically, $\sum_{i=1}^{n} i = n * i$. Thus, we can avoid subtracting ${}^{1}/_{a}$ each iteration, and instead subtract $x * {}^{1}/_{a}$ from the sum, where x is the number of results within the cutoff.

Finally, a optimization involved taking the two-dimensional `coords` array and converting it into a one-dimensional array of structs. The hope here was to imporve cache hits and reduce misses.

These tests were run on node granger3.

3 Observations

Some optimizations had nearly no effect on processing time, while others substantially contributed to the program's efficiency.

Compiler flags	Optimization	Time to calculate E (s)	Total Time (s)	Result
None		2.3691	2.3775	58261662.7646147534
-O1		1.0806	1.0893	58261662.7646147534
-O2		0.9236	0.9318	58261662.7646147534
-O3		0.9155	0.9232	58261662.7646147534
-Ofast		0.7045	0.7130	58261662.7646147460
-O3	For-loops start from 0	0.9620	0.9695	58261662.7646147534
-O3	Re-aligned for-loops	0.8882	0.8961	58261662.7646147534
-O3	Bring out dereferences	0.8993	0.9082	58261662.7646147534
-O3	Written-out squaring	0.9291	0.9380	58261662.7646147534
-O3	Combining exponents	0.6981	0.7071	58261662.7646147460
-O3	Comparing $r_{i,j}^2$ to cut^2	0.8461	0.8547	58261654.5481551588
-O3	Subtract $x * {}^1/_a$ later	1.1425	1.1512	58261662.7646149173
-O3	Convert 2D array to 1D struct array	0.8772	0.8853	58261662.7646147534
-O3	All above	0.5425	0.5521	58261654.5481553227
-O3 -ffast-math	All above	0.5132	0.5223	58261654.5481553227
-Ofast	All above	0.5104	0.5188	58261654.5481553227
-Ofast -fno-stack-protector	All above	0.4971	0.5047	58261654.5481553227

4 Analysis

The first notable observation is the magnitude of time difference just the first -O1 flag provides. Also notable is the insignificant difference between -O3 and -O2 compilation flags on the original program. -O3 has at best no effect, likely a slight adverse effect, compared to -O2.

Next, -Ofast on the original program produces not only a faster program, but also a slightly different result. The documentation indicates that -Ofast produces code that acts in a different manner from the IEEE specification, and that fact is apparent here.

Some optimizations resulted in a increase of time taken. This may be because those optimizations work better with other optimizations, such as in writing out the squaring as opposed to using the `pow()` method. There is also a difference in the produced result on the change to checking $r_{i,j}^2$ against cut^2. The best explanation is the gained resolution from not using the `sqrt()` function. Since `sqrt()` is normally implemented through an approximation requiring many iterations of floating-point operations, resolution is likely lost through the `sqrt()`.

Removing one of the buffer overflow defenses, checking for stack smashing, seems to produce a small time advantage. In a practical implementation, this may or may not be a preferrable optimization since it cuts out two-thirds of the protections against buffer overflow strikes, but it is interesting to see that it does provide a small time reduction. This optimization may prove to be more effective if there were more user-written functions called to during the execution of the program.

5 Conclusions

-O1 provides the majority of the compiler-level optimizations in this situation, and marginal improvements to execution time are added by -O2 or -O3. It is possible that some of the adverse affects of -O3 versus -O2 are due to general optimizations that would, on average, be effective optimizations. -Ofast enables -ffast-math and, thus, many unsafe but faster optimizations to the program. In addition, combining -O2 and -ffast-math is not faster than -Ofast, despite -Ofast is defined as -O3 -ffast-math when -O2 is when than -O3. This is perhaps because -O3 has optimizations that work better for -ffast-math than -O2.

In general, making the program do less computations results in less execution time, but can result in varrying mathematical results. Varrying optimizations can have varrying magnitudes of change in the calculated result.

Effective optimizations the programmer can take include reducing calls to heavy math functions, reducing dereferences to arrays, and putting off calculations until later when it is mathematically sound to do so and results in less computations.

6 Pledge

On my honor, I pledge that I did not give nor receive help on this assignment.

Louis Thomas

```c
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
  ! Bad coding example 1
  ! !
  ! Shamefully written by Ross Walker (SDSC, 2006)
  !
  ! This code reads a series of coordinates and charges from the file
  ! specified as argument $1 on the command line.
  !
  ! This file should have the format:
  !  I9
  ! 4F10.4   (repeated I9 times representing x,y,z,q)
  !
  ! It then calculates the following fictional function:
  !
  !              exp(rij*qi)*exp(rij*qj)   1
  !    E = Sum( ----------------------- - - )  (rij <= cut)
  !        j<i           r(ij)            a
  !
  ! where cut is a cut off value specified on the command line ($2),
  ! r(ij) is a function of the coordinates read in for each atom and
  ! a is a constant.
  !
  ! The code prints out the number of atoms, the cut off, total number of
  ! atom pairs which were less than or equal to the distance cutoff, the
  ! value of E, the time take to generate the coordinates and the time
  ! taken to perform the calculation of E.
  !
  ! All calculations are done in double precision.
  !+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
double **alloc_2D_double(int nrows, int ncolumns);
void double_2D_array_free(double **array);

/* struct coord
 *
 * This struct is aimed at reducing cache misses during execution.
 * a, b, and c correspond to coord[0], coord[1], and coord[2], respectively.
 */
typedef struct coord_t {
        double a, b, c;
} coord;

int main(int argc, char *argv[])
{
        long natom, i, j;
        long cut_count;

        /* Timer variables */
        clock_t time0, time1, time2;

        double cut;     /* Cut off for Rij in distance units */
        coord *coords; // -> changed to a 1D array of coord structs
        double *q;
        double total_e, current_e, vec2, rij;
        double a;
        FILE *fptr;
        char *cptr;

        a = 3.2;

        time0 = clock(); /*Start Time*/
        printf("Value of system clock at start = %ld\n",time0);

        /* Step 1 - obtain the filename of the coord file and the value of
           cut from the command line.
           Argument 1 should be the filename of the coord file (char).
           Argument 2 should be the cut off (float). */
        /* Quit therefore if iarg does not equal 3 = executable name,
           filename, cut off */
        if (argc != 3)
        {
                printf("ERROR: only %d command line options detected", argc-1);
                printf (" - need 2 options, filename and cutoff.\n");
                exit(1);
        }
        printf("Coordinates will be read from file: %s\n",argv[1]);

        /* Step 2 - Open the coordinate file and read the first line to
           obtain the number of atoms */
        if ((fptr=fopen(argv[1],"r"))==NULL)
        {
                printf("ERROR: Could not open file called %s\n",argv[1]);
                exit(1);
        }
        else
        {
                fscanf(fptr, "%ld", &natom);
        }

        printf("Natom = %ld\n", natom);

        cut = strtod(argv[2],&cptr);
        printf("cut = %10.4f\n", cut);

        /* Step 3 - Allocate the arrays to store the coordinate and charge
           data */
        // now allocate array of structs
        coords = (coord*)malloc(sizeof(*coords)*natom);
        if ( coords==NULL )
        {
                printf("Allocation error coords");
                exit(1);
        }
        q=(double *)malloc(natom*sizeof(double));
        if ( q == NULL )
        {
                printf("Allocation error q");
                exit(1);
        }

        /* Step 4 - read the coordinates and charges. */
        for (i = 0; i<natom; ++i)
        {
                // we have to read into the 1d array, now
                fscanf(fptr, "%lf %lf %lf %lf",
                                &(coords[i].a), &(coords[i].b),
```

```c
                                &(coords[i].c), &q[i]);
        }

        time1 = clock(); /*time after file read*/
        printf("Value of system clock after coord read = %ld\n",time1);


        /* Step 5 - calculate the number of pairs and E. - this is the
           majority of the work. */
        total_e = 0.0;
        cut_count = 0;

        // coordiX correspond to coords[X][i] -> coords[i].X
        double coordia;
        double coordib;
        double coordic;

        // this corresponds to q[i]
        double q_i;

        // this is the square of the cutoff to compare with vec2
        double cut2 = cut * cut;

        for (i = 0; i < natom; ++i)
        {
                // load derefernces here
                coordia = coords[i].a;
                coordib = coords[i].b;
                coordic = coords[i].c;

                q_i = q[i];
                for (j = 0; j < i; ++j)
                {
                        // now we use a literal square with new dereferences
                        vec2 = (coordia-coords[j].a)*(coordia-coords[j].a)
                                +(coordib-coords[j].b)*(coordib-coords[j].b)
                                +(coordic-coords[j].c)*(coordic-coords[j].c);
                        /* X^2 + Y^2 + Z^2 */
                        /* Check if this is below the cut off */

                        // we moved the sqrt inside the if then action and
                        // now compare to cut^2
                        if ( vec2 <= cut2 )
                        {
                                rij = sqrt(vec2); // <- moved here
                                /* Increment the counter of pairs below cutoff */
                                ++cut_count;
                                // now we add the multiples of the exponents in one
                                // exp usage
                                current_e = exp(rij*(q_i+q[j]))/rij;
                                // moved - 1.0/a; until after all the for loops
                                total_e = total_e + current_e;
                        }
                } /* for j=1 j<=natom */
        } /* for i=1 i<=natom */

        // moved here, fixed for not being calculated cut_count times
        total_e -= cut_count / a;

        time2 = clock(); /* time after reading of file and calculation */
        printf("Value of system clock after coord read and E calc = %ld\n",
                        time2);

        /* Step 6 - write out the results */
        printf("                           Final Results\n");
        printf("                           -------------\n");
        printf("                     Num Pairs = %ld\n",cut_count);
        printf("                       Total E = %14.10f\n",total_e);
        printf("     Time to read coord file = %14.4f Seconds\n",
                        ((double )(time1-time0))/(double )CLOCKS_PER_SEC);
        printf("         Time to calculate E = %14.4f Seconds\n",
                        ((double )(time2-time1))/(double )CLOCKS_PER_SEC);
        printf("        Total Execution Time = %14.4f Seconds\n",
                        ((double )(time2-time0))/(double )CLOCKS_PER_SEC);

        /* Step 7 - Deallocate the arrays - we should strictly check the
           return values here but for the purposes of this tutorial we can
           ignore this. */
        free(q);
        //double_2D_array_free(coords);
        // now we just allocate the 1d array like normal
        free(coords);

        fclose(fptr);

        exit(0);
}

double **alloc_2D_double(int nrows, int ncolumns)
{
        /* Allocates a 2d_double_array consisting of a series of pointers
           pointing to each row that are then allocated to be ncolumns
           long each. */

        /* Try's to keep contents contiguous - thus reallocation is
           difficult! */

        /* Returns the pointer **array. Returns NULL on error */
        int i;

        double **array = (double **)malloc(nrows*sizeof(double *));
        if (array==NULL)
                return NULL;
        array[0] = (double *)malloc(nrows*ncolumns*sizeof(double));
        if (array[0]==NULL)
                return NULL;

        for (i = 1; i < nrows; ++i)
                array[i] = array[0] + i * ncolumns;

        return array;

}

void double_2D_array_free(double **array)
{
        /* Frees the memory previously allocated by alloc_2D_double */
        free(array[0]);
        free(array);
}
```

241: